A Verified Simple Prover for First-Order Logic

Villadsen, Jørgen; Schlichtkrull, Anders; From, Andreas Halkjær

Published in:
Proceedings of the 6th Workshop on Practical Aspects of Automated Reasoning (PAAR)

Publication date:
2018

Document Version
Publisher's PDF, also known as Version of record

A Verified Simple Prover for First-Order Logic

Jørgen Villadsen, Anders Schlichtkrull, and Andreas Halkjær From

DTU Compute, AlgoLoG, Technical University of Denmark, 2800 Kongens Lyngby, Denmark
jovi@dtu.dk

Abstract. We present a simple prover for first-order logic with certified soundness and completeness in Isabelle/HOL, taking formalizations by Tom Ridge and others as the starting point, but with the aim of using the approach for teaching logic and verification to computer science students at the bachelor level. The prover is simple in the following sense: It is purely functional and can be executed with rewriting rules or as code generation to a number of functional programming languages. The prover uses no higher-order functions, that is, no function takes a function as argument or returns a function as its result. This is advantageous when students perform rewriting steps by hand. The prover uses the logic of first-order logic on negation normal form with a term language consisting of only variables. This subset of the full syntax of first-order logic allows for a simple proof system without resorting to the much weaker propositional logic.

1 Introduction

Our motivation is in some ways similar to the development of lean theorem provers like leanTAP and leanCoP [4], although with more focus on the use in teaching and less focus on reaching considerable performance. In contrast to leanTAP and leanCoP we also require formal verification of the soundness and completeness of the prover, in our case in Isabelle/HOL [3]. The development described in this paper is available online:

https://bitbucket.org/isafol/isafol/src/master/Simple_Prover/

Let us recall the following conclusions concerning leanCoP [4, p. 159]:

> Lean provers can easily be integrated and modified by people who do not have a deep knowledge about fully automatic provers. Since it is much easier (and faster) to understand a few lines of Prolog code than several thousand lines of, e.g., C code, lean theorem provers are also very well suited for teaching purposes. Finally, for the same reason it is also much easier to verify completeness and correctness of lean theorem provers.

Using ISO Prolog the compact code for leanCoP amounts to the following clauses:

```
prove(M,I) :- append(Q,[C|R],M), \+member(-_,C),
  append(Q,R,S), prove([!],[[-!|C]|S],[],I).
prove([],_,_,_).
prove([L|C],M,P,I) :- (-N=L; -L=N) -> (member(U,P),
  unify_with_occurs_check(U,N);
  append(Q,[D|R],M), copy_term(D,E), append(A,[U|B],E),
  unify_with_occurs_check(U,N),
  append(A,B,F), (D==E -> append(R,Q,S); length(P,K), K<I,
  append(R,[D|Q],S)), prove(F,S,[L|P],I)), prove(C,M,P,I).
```

We agree with the authors that the leanCoP code is well suited for teaching and in particular more so than thousands of lines of C code. Clearly, the concise code of leanCoP is much easier to get an overview of than a huge C program. Already in the representation of first-order formulas leanCoP shines in that it can simply rely on the terms of Prolog. However, the conciseness and simple representation does, in our opinion, come at a price. The code of leanCoP is written in the Prolog language and our experience from teaching both C and Prolog is that our students understand the execution model of C much faster than that of Prolog which involves backtracking and unification without occurs check. On top of that, leanCoP relies on some rather advanced features that go beyond pure Prolog to manipulate terms and to get the execution right. In particular it relies on ISO Prolog's build-in unification predicate with occurs-check `unify_with_occurs_check/2`, the `copy_term/2` predicate, the special equivalence predicate `==/2` and the if-then-else predicate whose semantics is actually surprisingly complex. We believe that there must be some middle ground between thousands of lines of C code and a Prolog program that relies on very complex features of the language.

Our simple prover is an attempt at reaching this middle ground. The prover is purely functional and can be executed with rewriting rules. We provide all rewriting rules, also for if-then-else constructs, arithmetic, list operations, etc. We avoid the more advanced features of functional programming such as higher-order functions, that is, no function takes a function as argument or returns a function as its result. This is advantageous when students perform rewriting steps by hand.

One of the advantages of leanCoP is that because of its small size it should be easy to verify its completeness. This does, however, require reasoning about the advanced Prolog features mentioned above, which is no easy task. In contrast functional programming allows equational reasoning, and our simple prover is formally verified in Isabelle/HOL. This also means that modifications of the code are possible with interactive feedback in the Isabelle proof assistant that indicates where the soundness and completeness arguments need to be adapted to the changes.

To simplify things our simple prover does not allow for complex terms — all terms are variables, with free variables functioning as constants, if necessary. We find this acceptable since it means that we can delay teaching unification to e.g. a more advanced course on automated reasoning.

We have taken the formalizations by Tom Ridge and others [5,6] as the starting point, but we have totally rewritten the program (with no higher-order functions) as well as the soundness and completeness proof (now using the Isar proof language [8]). We discuss this further in section 7 on related work.

Section 2 explains the proof system on which the prover is based. Section 3 explains the prover as encoded in Isabelle — in particular the syntax and semantics of first-order logic as well as the prover and its execution on a small example. Section 4 explains the formalized soundness proof. Section 5 explains the formalized completeness proof. Section 6 considers code generation from Isabelle/HOL to functional programming languages. Section 7 is a discussion of related work. Finally, section 8 concludes on the paper.

2 The Proof System

We consider essentially the same proof system as Tom Ridge and others [5,6] with a sequent as an ordered list of formulas in negation normal form (nnf), cf. Fig. 1. We will always use the rules in a backward chaining fashion — to prove a sequent we choose a rule or axiom whose conclusion fits the sequent and then prove its premises recursively in the same way. We choose a discipline where we always apply Ax and $\overline{Ax}$ rather than $NoAx$ and $\overline{NoAx}$ whenever this is possible. With this discipline the described backward chaining procedure is deterministic.

The rules Ax and $\overline{Ax}$ denote the leaves of the derivation tree, where the complement of the considered predicate occurs at a later position in the sequent, producing a tautology. The rules $NoAx$ and $\overline{NoAx}$ apply when the complement does not appear in the sequent and move the predicate to the end of the sequent to continue work on the rest of the formulas.

Since a sequent already denotes a disjunction, the $\vee$-rule breaks down a disjunction into its parts and adds them to the end of the sequent. For conjunctions however, the derivation tree branches using the $\wedge$-rule to attempt a derivation of both conjuncts separately.

In the $\exists$-rule for existential quantifiers, we use a superscript, initially 0, to denote the next variable to try to use as witness for the existential. The quantified formula is instantiated with this variable in the premise, but we also add the original formula with an incremented superscript behind its instantiation. Thus all witnesses will eventually be tried.

Finally, the premise of the $\forall$-rule, for universal quantifiers, is the formula instantiated with a fresh variable. The idea is that if the formula holds for this arbitrary variable, it must hold for any variable.

Note that for invalid formulas the derivation will not terminate as some branch will never end in a leaf.

Rule	Note
$\dfrac{}{\vdash P(v_{i_1},...,v_{i_k}), \Gamma, \overline{P}(v_{i_1},...,v_{i_k}), \Delta}\ Ax$	Leaf of the derivation tree.
$\dfrac{}{\vdash \overline{P}(v_{i_1},...,v_{i_k}), \Gamma, P(v_{i_1},...,v_{i_k}), \Delta}\ \overline{Ax}$	Leaf of the derivation tree.
$\dfrac{\vdash \Gamma, P(v_{i_1},...,v_{i_k})}{\vdash P(v_{i_1},...,v_{i_k}), \Gamma}\ NoAx$	
$\dfrac{\vdash \Gamma, \overline{P}(v_{i_1},...,v_{i_k})}{\vdash \overline{P}(v_{i_1},...,v_{i_k}), \Gamma}\ \overline{NoAx}$	
$\dfrac{\vdash \Gamma, A, B}{\vdash A \vee B, \Gamma}\ \vee$	
$\dfrac{\vdash \Gamma, A \qquad \vdash \Gamma, B}{\vdash A \wedge B, \Gamma}\ \wedge$	The only branching rule.
$\dfrac{\vdash \Gamma, [v_i/x]A, (\exists x.A)^{i+1}}{\vdash (\exists x.A)^i, \Gamma}\ \exists$	Superscripts are only relevant for this rule, and allow $[v_i/x]A$ to be instantiated for all i.
$\dfrac{\vdash \Gamma, [v_r/x]A}{\vdash \forall x.A, \Gamma}\ \forall$	v_r is a fresh free variable, chosen as $r = max(S) + 1$, where S is the set of subscripts already used for the free variables in A ($r = 0$ if there are no free variables in A).

Fig. 1. Proof System

3 The Simple Prover

The proof system is encoded in Isabelle. We present the syntax, semantics and definition of the prover before showing the progression of a small example proof.

3.1 Syntax

The datatype *nnf* represents formulas in negation normal form. Variables are represented using de Bruijn indices, the type of predicate names is *id* and the *bool* in the predicate constructor *Pre* is a sign — when it is *True* we have our hands on an atom and when it is *False* the negation of an atom.

type-synonym *id* = *nat*

datatype *nnf* =
  Pre bool id ‹*nat list*› | *Con nnf nnf* | *Dis nnf nnf* | *Uni nnf* | *Exi nnf*

3.2 Test example

TEST P Q abbreviates a test formula which is proved in Isabelle and converted into negation normal form.

abbreviation (*input*) *TEST P Q* ≡ (∃ *x*. *P x* ∨ *Q x*) ⟶ (∃ *x*. *Q x*) ∨ (∃ *x*. *P x*)

proposition *TEST P Q*

proposition *TEST P Q* = (∀ *x*. ¬ *P x* ∧ ¬ *Q x*) ∨ (∃ *x*. *Q x*) ∨ (∃ *x*. *P x*)

The definition *test* is the same formula in our formalization.

abbreviation (*input*) *P-id* ≡ *0*

abbreviation (*input*) *Q-id* ≡ *Suc 0*

definition
  test ≡ *Dis*
    (*Uni* (*Con* (*Pre False P-id* [*0*]) (*Pre False Q-id* [*0*])))
    (*Dis* (*Exi* (*Pre True Q-id* [*0*])) (*Exi* (*Pre True P-id* [*0*])))

3.3 Semantics

For the semantics we, for teaching purposes, consider only countable universes using unit lists (an arbitrary universe is too abstract while a universe of natural numbers may fool students to think we only care about numbers). It is not difficult to use another infinite type than that of unit lists; this can be done by, for instance, declaring an arbitrary infinite type — this is the approach used

by Ridge [5,6] — or using HOL's type *ind* of individuals — as suggested by Ridge in a footnote [6]. The change can be made locally and does not affect the completeness proof. Note that *proxy* is not the universe but the set from which the elements of the universes are taken.

type-synonym *proxy* = *unit list*

type-synonym *model* = *proxy set* × (*id* ⇒ *proxy list* ⇒ *bool*)

type-synonym *environment* = *nat* ⇒ *proxy*

definition *is-model-environment* :: *model* ⇒ *environment* ⇒ *bool* **where**
is-model-environment m e ≡ ∀ *n*. *e n* ∈ *fst m*

primrec *semantics* :: *model* ⇒ *environment* ⇒ *nnf* ⇒ *bool* **where**
semantics m e (*Pre b i v*) = (*b* = *snd m i* (*map e v*)) |
semantics m e (*Con p q*) = (*semantics m e p* ∧ *semantics m e q*) |
semantics m e (*Dis p q*) = (*semantics m e p* ∨ *semantics m e q*) |
semantics m e (*Uni p*) =
(∀ *z* ∈ *fst m*. *semantics m* (λ*x*. *case x of 0* ⇒ *z* | *Suc n* ⇒ *e n*) *p*) |
semantics m e (*Exi p*) =
(∃ *z* ∈ *fst m*. *semantics m* (λ*x*. *case x of 0* ⇒ *z* | *Suc n* ⇒ *e n*) *p*)

The definition *is-model-environment* is used later together with *semantics* to state the soundness and completeness theorem.

3.4 Prover

In Isabelle, the prover is defined as a number of functions by their defining equations in a style similar to the ML functional programming languages. From these we can immediately prove a set of lemmas repeating the equations to illustrate to our students that these are building blocks that can be reasoned about like any other lemmas of Isabelle. These lemmas can be taken as rewrite rules that students can study and hand-run to see how the prover works.

The prover uses no higher-order functions, that is, no function takes a function as argument or returns a function as its result. Except for *prover* all rewrite rules are primitive recursive. We represent sequents as lists of formulas where each formula is tagged by the superscript used for the ∃-rule.

The functions *prover*, *solves* and *solve* drive the main function *track* by trying to prove all the given branches, represented as a list of sequents.

We use *check* to start this with just a single branch containing the given formula.

lemma *check p* ≡ *prover* [[(*0*,*p*)]]

lemma *prover* (*h* # *t*) ≡ *prover* (*solves* (*h* # *t*))
lemma *prover* [] ≡ *True*

lemma *solves* [] ≡ []
lemma *solves* (*h* # *t*) ≡ *solve h* @ *solves t*

lemma *solve* [] ≡ [[]]
lemma *solve* (*h* # *t*) ≡ *track t* (*fst h*) (*snd h*)

lemma *track s n* (*Pre b i v*) ≡ *stop* [*s* @ [(*0*,*Pre b i v*)]] (*Pre* (¬ *b*) *i v*) (*base s*)
lemma *track s n* (*Con p q*) ≡ [*s* @ [(*0*,*p*)],*s* @ [(*0*,*q*)]]
lemma *track s n* (*Dis p q*) ≡ [*s* @ [(*0*,*p*),(*0*,*q*)]]
lemma *track s n* (*Uni p*) ≡ [*s* @ [(*0*,*subst 0* (*fresh* (*frees* (*Uni p* # *base s*))) *p*)]]
lemma *track s n* (*Exi p*) ≡ [*s* @ [(*0*,*subst 0 n p*),(*Suc n*,*Exi p*)]]

The main function *track* implements the proof rules from the conclusion to the premises. The case for conjunction creates two new branches, while a disjunction extends the current branch as expected. For universal quantifiers, the variable is replaced by a fresh constant using functions described below while for existential quantifiers, it checks if the variable n satisfies the quantified formula and also re-tags the existential with the successor of n, in case it does not. In the predicate case of the main function *track* it determines whether we are in a leaf using *stop*.

lemma *stop c p* [] ≡ *c*
lemma *stop c p* (*h* # *t*) ≡ (*if p* = *h then* [] *else stop c p t*)

We use *stop* to check if a formula exists in the given list and in that case return an empty list of new goals, thus stopping the search on that branch. Otherwise *stop* returns the given default.

The function *base* removes the superscript tag from each formula in a sequent, returning a list of formulas.

lemma *base* [] ≡ []
lemma *base* (*h* # *t*) ≡ *snd h* # *base t*

The function *subst* takes care of substitution in a formula. Since we only have variables in our term language, the given s will always be a variable.

lemma *subst x s* (*Pre b i v*) ≡ *Pre b i* (*mend x s v*)
lemma *subst x s* (*Con p q*) ≡ *Con* (*subst x s p*) (*subst x s q*)
lemma *subst x s* (*Dis p q*) ≡ *Dis* (*subst x s p*) (*subst x s q*)
lemma *subst x s* (*Uni p*) ≡ *Uni* (*subst* (*Suc x*) (*Suc s*) *p*)
lemma *subst x s* (*Exi p*) ≡ *Exi* (*subst* (*Suc x*) (*Suc s*) *p*)

This makes the quantifier cases easy to handle when using de Bruijn indices, as incrementing the term to account for passing a quantifier is just incrementing the variable. In the predicate case we use the function *mend* which applies *more* to each argument of the predicate after subtracting the variable we are substituting for, x, from it. This way, *more* can match on the result of the subtraction

to determine the variable's index relative to what we are substituting for. If the result is positive, this is not the variable to substitute for and it is decremented instead to account for the removed quantifier. If *sub x h* is zero, then *more* delegates to *over* with the subtraction the other way around. Thus if *over* receives a zero, the variable is equal to what we are substituting for and *over* returns the *s* we are substituting with, otherwise the variable is smaller and returned as is.

lemma *mend x s* [] ≡ []
lemma *mend x s* (*h* # *t*) ≡ *more x s h* (*sub h x*) # *mend x s t*

lemma *more x s h 0* ≡ *over s h* (*sub x h*)
lemma *more x s h* (*Suc n*) ≡ *dec h*

lemma *over s h 0* ≡ *s*
lemma *over s h* (*Suc n*) ≡ *h*

The function *fresh* returns a new variable given a list of used variables (we use subtraction and then addition to find the maximum of two natural numbers).

lemma *fresh* [] ≡ *0*
lemma *fresh* (*h* # *t*) ≡ *Suc* (*add* (*sub* (*dec* (*fresh t*)) *h*) *h*)

The function *free* returns the free variables in a formula, taking into account the binding of variables by the universal and existential quantifiers. The latter is accomplished by the functions *dump* and *dash* which together decrement all variables in the list while simultaneously removing any zeros, as these are bound by the quantifier and thus not free.

lemma *frees* [] ≡ []
lemma *frees* (*h* # *t*) ≡ *free h* @ *frees t*

lemma *free* (*Pre b i v*) ≡ *v*
lemma *free* (*Con p q*) ≡ *free p* @ *free q*
lemma *free* (*Dis p q*) ≡ *free p* @ *free q*
lemma *free* (*Uni p*) ≡ *dump* (*free p*)
lemma *free* (*Exi p*) ≡ *dump* (*free p*)

lemma *dump* [] ≡ []
lemma *dump* (*h* # *t*) ≡ *dash* (*dump t*) *h*

lemma *dash l 0* ≡ *l*
lemma *dash l* (*Suc n*) ≡ *n* # *l*

For auxiliary functions we prefer short, single-word names (at most 5 characters) that describe the functionality as well as possible within these restrictions. Given this preference we believe the names work well, but one could argue that they are a little far-fetched. If teachers or students disagree with our preference they can fortunately easily rename the functions; renaming and other minor

modifications are relatively easy in Isabelle and the formal proofs ensure that doing so does not introduce an accidental mistake.

In addition to these rewriting rules there are also some data and library rewrite rules. See the Appendix. The data rewrite rules state equalities and inequalities based on the datatypes for the formulas. The library rewriting rules state properties of auxiliary concepts such as *if-then-else* and the natural numbers.

We state the soundness and completeness theorem. The proof is about one thousand lines and described in the following sections.

theorem *check p* = (∀ *m e*. *is-model-environment m e* ⟶ *semantics m e p*)

3.5 Test example continued

The following proposition shows that our prover *check* proves the test formula *test*.

proposition *check test*
unfolding *test-def*
unfolding *program*(*1*)
unfolding *program*(*2*)
unfolding *program*(*3*−) *data library*
unfolding *program*(*2*)
unfolding *program*(*3*−) *data library*
unfolding *program*(*2*)
unfolding *program*(*3*−) *data library*
unfolding *program*(*2*)
unfolding *program*(*3*−) *data library*
unfolding *program*(*2*)
unfolding *program*(*3*−) *data library*
unfolding *program*(*2*)
unfolding *program*(*3*−) *data library*
unfolding *program*(*2*)
unfolding *program*(*3*−) *data library*
by (*rule TrueI*)

We show the result of every other unfolding in Fig. 2, namely the situation after each *program*(*2*) line where the *prover* function is unfolded. The single *program*(*1*) line simply unfolds the *check* function.

4 Soundness

We prove soundness of the prover. First we model the derivation of a sequent and show that finite derivations correspond to executions of the prover. Now all that remains is to show soundness of the finite derivations.

```
check
  (Dis (Uni (Con (Pre False 0 [0]) (Pre False (Suc 0) [0])))
    (Dis (Exi (Pre True (Suc 0) [0])) (Exi (Pre True 0 [0]))))

prover
  [[(0, Dis (Uni (Con (Pre False 0 [0]) (Pre False (Suc 0) [0])))
          (Dis (Exi (Pre True (Suc 0) [0]))
            (Exi (Pre True 0 [0]))))]]

prover
  [[(0, Uni (Con (Pre False 0 [0]) (Pre False (Suc 0) [0]))),
    (0, Dis (Exi (Pre True (Suc 0) [0])) (Exi (Pre True 0 [0])))]]

prover
  [[(0, Dis (Exi (Pre True (Suc 0) [0])) (Exi (Pre True 0 [0]))),
    (0, Con (Pre False 0 [0]) (Pre False (Suc 0) [0]))]]

prover
  [[(0, Con (Pre False 0 [0]) (Pre False (Suc 0) [0])),
    (0, Exi (Pre True (Suc 0) [0])), (0, Exi (Pre True 0 [0]))]]

prover
  [[(0, Exi (Pre True (Suc 0) [0])), (0, Exi (Pre True 0 [0])),
    (0, Pre False 0 [0])],
   [(0, Exi (Pre True (Suc 0) [0])), (0, Exi (Pre True 0 [0])),
    (0, Pre False (Suc 0) [0])]]

prover
  [[(0, Exi (Pre True 0 [0])), (0, Pre False 0 [0]),
    (0, Pre True (Suc 0) [0]),
    (Suc 0, Exi (Pre True (Suc 0) [0]))],
   [(0, Exi (Pre True 0 [0])), (0, Pre False (Suc 0) [0]),
    (0, Pre True (Suc 0) [0]),
    (Suc 0, Exi (Pre True (Suc 0) [0]))]]

prover
  [[(0, Pre False 0 [0]), (0, Pre True (Suc 0) [0]),
    (Suc 0, Exi (Pre True (Suc 0) [0])), (0, Pre True 0 [0]),
    (Suc 0, Exi (Pre True 0 [0]))],
    [(0, Pre False (Suc 0) [0]), (0, Pre True (Suc 0) [0]),
    (Suc 0, Exi (Pre True (Suc 0) [0])), (0, Pre True 0 [0]),
    (Suc 0, Exi (Pre True 0 [0]))]]

True
```

Fig. 2. Prover steps for the test example

Derivations are modeled as an inductive set of derived sequents. We call such a set the *calculation*:

inductive-set *calculation* :: ‹*sequent* ⇒ (*nat* × *sequent*) *set*› **for** *s* **where**
‹(*0*,*s*) ∈ *calculation s*› |
‹(*n*,*k*) ∈ *calculation s* ⟹ *l* ∈ *set* (*solve k*) ⟹ (*Suc n*,*l*) ∈ *calculation s*›

Each sequent is labeled with the level at which it is added to the set, making it simpler to prove properties of the proof system inductively. Note that it represents the derivation backwards, adding the leaves as the final step, and that *calculation* tracks all branches simultaneously.

Our corollary *finite-calculation-check* states that *check*, which represents a terminating execution of the prover, is true exactly for those formulas whose derivations are finite:

corollary *finite-calculation-check*: ‹*finite* (*calculation* (*make-sequent* [*p*])) = *check p*›

Its proof establishes a connection between the labels in a *calculation* and the number of iterations of the solver, showing that if the *calculation* is finite, the solver will at one point stop producing new goals and terminate, and vice versa.

The soundness theorem for derivations is

lemma *soundness*:
assumes ‹*finite* (*calculation* (*make-sequent s*))›
shows ‹*valid s*›

In its proof we show that every derived sequent is valid by considering them in opposite order of how they were added. The calculation is finite, so there must exist some largest label m. Any sequent labeled m is a leaf, as otherwise it would have been derived from a new sequent (the premise) with higher label. Thus it must be valid since leaves in the proof system are trivially valid. In the cases where the label is smaller than m we can assume by induction that the premises of the added sequent are valid, as these are one step closer to the leaves. We then prove validity for each case of the considered formula in the sequent in a similar fashion to a soundness proof for natural deduction rules. Take for instance the case for conjunction where we break *Con p q* # *s* down to two new branches, *p* # *s* and *q* # *s*. These new sequents have higher labels so we can assume the validity of those and have to prove the validity of the sequent with the conjunction. Finally, by induction, only valid sequents appear in a finite *calculation*.

Together with the lemma above, this gives us the soundness direction of the correctness theorem at the end of section 3.

5 Completeness

For completeness we still consider the *calculation* described above, this time focusing on infinite derivation attempts. We extract a counter-model to the initial formula from an infinite, so-called *failing*, path in the attempted derivation, proving its invalidness. By contraposition, any valid formula has a finite derivation which the prover will find, and thus the prover is complete. The existence of a failing path follows from König's lemma, but we follow the same approach as Ridge [5,6] and do not explicitly invoke the lemma, but instead define a function *failing-path* which given an infinite calculation finds an infinite path in it, and then when given a natural number returns the nth node in the infinite path. The function uses Hilbert's choice operator to pick nodes in the infinite part of the derivation tree.

For an example of a failing path consider the following derivation which also constitutes a failing path:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\vdash q(v_1), \neg p(v_0), \neg p(v_1), (\exists x.\ \neg p(x))^2}}{\vdash (\exists x.\ \neg p(x))^1, q(v_1), \neg p(v_0)}\exists \quad \dfrac{}{NoAx}}{\vdash \neg p(v_0), (\exists x.\ \neg p(x))^1, q(v_1)}NoAx}{\vdash q(v_1), \neg p(v_0), (\exists x.\ \neg p(x))^1}\exists}{\vdash (\exists x.\ \neg p(x))^0, q(v_1)}\vee}{\vdash (\exists x.\ \neg p(x)) \vee q(v_1)}$$

We model failing paths as functions from natural numbers into sequents, i.e. if f is a failing path in derivation d then $f\ 0$ is the first sequent on the infinite path, $f\ 1$ is the second and so on. The function *failing-path* therefore takes a derivation and returns a failing path in it.

We say that a (failing) path *contains* a formula at label n if the formula appears in a sequent at level n on the path, and that it *considers* the formula at label n if it appears at the head of the sequent on level n on the path. Since a failing path is infinite and our proof system always removes a formula from the front of a sequent and adds new premises to the end, any formula contained on the path will eventually be considered on the path because it is rotated to the front. Next we describe how each type of formula is propagated along a failing path. For all types it is the case that if the formula is contained in a failing path, then a set of instances of its subformulas is eventually added and contained in all the following levels of the path. For predicates the instance of a subformula is the formula itself, but for all other types instances of proper subformulas are added: Conjunctions have one of their conjuncts added, disjunctions have both their disjuncts added and quantified formulas have instances of the formula quantified over added. Any existential must initially have been tagged with zero, as none of the other rules increment the superscript. This allows us to disregard the superscript when considering propagation.

Our counter-model consists of a universe and predicate denotation and is constructed from a failing path in the following way: The universe is the universe of unit lists as above and an atom is true if and only if it does not appear on a failing path. In the above example, the counter-model therefore considers $q(v_1)$ false and any other atom true. We show by induction on the size of a formula that any formula contained on the path is falsified by the model given the propagation properties sketched above. Finally the initial formula is falsified since this is the first formula on the path and thus our counter-model is actually a counter-model. By reusing the connection between the prover and a *calculation* established in section 4, we can transfer this result to the prover, giving us the completeness part of the correctness theorem at the end of section 3.

6 Code Generation

As mentioned, the prover can be exported to a number of functional languages using code generation. Isabelle allows exporting a large subset of its higher-order logic to code and thus in general it was easy to keep our prover functions within this subset. The possibility of code generation means that the prover can be integrated into other developments.

Our prover is expressed in terms of a certain *iterator* function for which we prove the following lemma:

iterator g f c = (*if g c then True else iterator g f* (*f c*))

It is then possible to execute the prover in the Isabelle formalization using the Standard ML extension Isabelle/ML and the *value* command:

value (*check test*)

Alternatively the generated code can be exported to Haskell, OCaml, Scala and Standard ML. We first discussed these possibilities in our preliminary informal paper entitled "Code Generation for a Simple First-Order Prover" presented at the Isabelle Workshop 2016.

Isabelle itself includes automated theorem provers integrated as tactics (called proof methods in Isabelle-lingo) which can automatically prove subgoals. It could be interesting to make such a tactic with formally verified completeness, however, it is not straight forward to do from our prover since any logical operation in Isabelle should then be made by calls into Isabelle's kernel.

7 Related Work

As mentioned, our starting point is the formalizations by Ridge and others [5,6]. The inspiration is clear, but we have totally rewritten the prover.

In particular, Ridge's formalization uses higher-order functions while we avoid these entirely. This is clearest in the definitions of substitution. Ridge does substitution by recursing on the formula datatype and passing down a

function from variables to variables. Whenever the recursion passes a quantifier, the function is adjusted to account for this, and when a predicate is reached the function is applied as is to the variables. We instead pass down exactly two variables (where one is to be replaced by the other) and then when a predicate is reached all the necessary adjustments to variables are made on the spot. Passing a quantifier in our recursion amounts to increasing the two variables by one.

Likewise, for the soundness and completeness proofs our starting point was the formalization by Ridge, but the proofs have been rewritten. That is, while the big picture of the proof is the same and many auxiliary definitions, lemmas and theorems are essentially the same, we have made many changes — the biggest one being that the proofs are now written in the structured, declarative Isar-style [8] instead of a procedural style with consecutive applications of tactics.

Except for the line of work by Ridge and others [5,6], we are not aware of formally verified, sound and complete, provers for first-order logic, cf. [7] and [1] (the latter reference does hint at a prover but the code is not provided as far as we can see).

Various proof systems have formally been proved sound and complete, cf. again [7] and [1], and recently even a higher-order prover has formally been proved sound [2] (but so far not complete).

8 Conclusion

We have presented a simple prover for first-order logic with certified soundness and completeness in Isabelle/HOL. We have taken the formalizations by Tom Ridge and others [5,6] as the starting point, but we have totally rewritten the program as well as the soundness and completeness proof. The formalization — with a couple of examples and extra features — is about 1900 lines including blank lines but excluding comments. All in all it takes around 5 seconds real time to verify on a fairly standard computer.

We have used the approach for teaching logic and verification to selected mathematics and computer science students at the bachelor level. The student can study and modify the formalization in Isabelle but we do not always consider the details in the soundness and completeness proofs. As a substantial project assignment we have asked the students to reduce the simple prover from first-order logic to just propositional logic.

Concerning regular courses, we intend to teach the prover to BSc students in software technology who do not necessarily have a very strong background in logic or discrete mathematics. The very simple approach to fairness and the lack of unification means that these students have a better chance of grasping the completeness proof. Furthermore, the proof is in spirit very similar to more advanced and abstract approaches to completeness such as the formalization of abstract soundness and completeness theorems due to Blanchette, Popescu and Traytel [1]. The most ambitious students could use our development as a springboard to understanding the abstract results.

The students can perform rewriting steps by hand or using Isabelle. The prover uses no higher-order functions. Code generation to a number of functional programming languages is also possible, and using Isabelle's code reflection to Standard ML, in particular the extension Isabelle/ML, it is fully integrated in the Isabelle formalization. Future work includes further polishing the proofs of soundness and completeness in order to make them easier to understand for students and researchers. Future work also includes experiments with a stand-alone tool for illustrating the unfolding rules.

Acknowledgements

We thank Tom Ridge, Uwe Waldmann, Alexander Birch Jensen, Andrea Dittadi, John Bruntse Larsen, Thomas Michaelsen Pethick, Camilla Lipczak Nielsen, Klara Emilie Elmegaard and the Isabelle team at TUM for help and discussions.

Appendix: Data and Library Rewrite Rules

The following data rewrite rules state equalities and inequalities based on the datatypes for the formulas.

lemma $Pre\ b\ i\ v = Con\ p\ q \equiv False$
lemma $Con\ p\ q = Pre\ b\ i\ v \equiv False$

lemma $Pre\ b\ i\ v = Dis\ p\ q \equiv False$
lemma $Dis\ p\ q = Pre\ b\ i\ v \equiv False$

lemma $Pre\ b\ i\ v = Uni\ p \equiv False$
lemma $Uni\ p = Pre\ b\ i\ v \equiv False$

lemma $Pre\ b\ i\ v = Exi\ p \equiv False$
lemma $Exi\ p = Pre\ b\ i\ v \equiv False$

lemma $Con\ p\ q = Dis\ p'\ q' \equiv False$
lemma $Dis\ p'\ q' = Con\ p\ q \equiv False$

lemma $Con\ p\ q = Uni\ p' \equiv False$
lemma $Uni\ p' = Con\ p\ q \equiv False$

lemma $Con\ p\ q = Exi\ p' \equiv False$
lemma $Exi\ p' = Con\ p\ q \equiv False$

lemma $Dis\ p\ q = Uni\ p' \equiv False$
lemma $Uni\ p' = Dis\ p\ q \equiv False$

lemma $Dis\ p\ q = Exi\ p' \equiv False$
lemma $Exi\ p' = Dis\ p\ q \equiv False$

lemma $Uni\ p = Exi\ p' \equiv False$
lemma $Exi\ p' = Uni\ p \equiv False$

lemma $Pre\ b\ i\ v = Pre\ b'\ i'\ v' \equiv b = b' \wedge i = i' \wedge v = v'$
lemma $Con\ p\ q = Con\ p'\ q' \equiv p = p' \wedge q = q'$
lemma $Dis\ p\ q = Dis\ p'\ q' \equiv p = p' \wedge q = q'$
lemma $Uni\ p = Uni\ p' \equiv p = p'$
lemma $Exi\ p = Exi\ p' \equiv p = p'$

The following library rewriting rules state properties of auxiliary concepts such as *if-then-else* as well as lists and natural numbers. They also state equalities and inequalities based on the datatypes of lists and natural numbers, among others.

lemma $add\ x\ 0 \equiv x$
lemma $add\ x\ (Suc\ n) \equiv Suc\ (add\ x\ n)$

lemma $sub\ x\ 0 \equiv x$
lemma $sub\ x\ (Suc\ n) \equiv dec\ (sub\ x\ n)$

lemma $dec\ 0 \equiv 0$
lemma $dec\ (Suc\ n) \equiv n$

lemma $[]\ @\ l \equiv l$
lemma $(h\ \#\ t)\ @\ l \equiv h\ \#\ t\ @\ l$

lemma $if\ True\ then\ x\ else\ y \equiv x$
lemma $if\ False\ then\ x\ else\ y \equiv y$

lemma $\neg\ True \equiv False$
lemma $\neg\ False \equiv True$

lemma $fst\ (x,y) \equiv x$
lemma $snd\ (x,y) \equiv y$

lemma $0 = 0 \equiv True$
lemma $[] = [] \equiv True$

lemma $True = True \equiv True$
lemma $False = False \equiv True$

lemma $True \wedge b \equiv b$
lemma $False \wedge b \equiv False$

lemma $0 = Suc\ n \equiv False$
lemma $Suc\ n = 0 \equiv False$

lemma $[] = h\ \#\ t \equiv False$
lemma $h\ \#\ t = [] \equiv False$

lemma $True = False \equiv False$
lemma $False = True \equiv False$

lemma $(x,y) = (x',y') \equiv x = x' \wedge y = y'$

lemma $Suc\ n = Suc\ n' \equiv n = n'$

lemma $h\ \#\ t = h'\ \#\ t' \equiv h = h' \wedge t = t'$

References

1. Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. Soundness and completeness proofs by coinductive methods. *Journal of Automated Reasoning*, 58(1):149–179, 2017.
2. Ramana Kumar, Rob Arthan, Magnus O. Myreen, and Scott Owens. Self-formalisation of higher-order logic: Semantics, soundness, and a verified implementation. *Journal of Automated Reasoning*, 56(3):221–259, 2016.
3. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
4. Jens Otten and Wolfgang Bibel. leanCoP: lean connection-based theorem proving. *Journal of Symbolic Computation*, 36:139–161, 2003.
5. Tom Ridge. A mechanically verified, efficient, sound and complete theorem prover for first order logic. *Archive of Formal Proofs*, September 2004. `http://isa-afp.org/entries/Verified-Prover.shtml`, Formal proof development.
6. Tom Ridge and James Margetson. A mechanically verified, sound and complete theorem prover for first order logic. In *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, pages 294–309, 2005.
7. Anders Schlichtkrull. Formalization of the resolution calculus for first-order logic. *Journal of Automated Reasoning*, (article not yet assigned to an issue) 1–30, 2018.
8. Markus Wenzel. Isar - A generic interpretative approach to readable formal proof documents. In *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings*, pages 167–184, 1999.